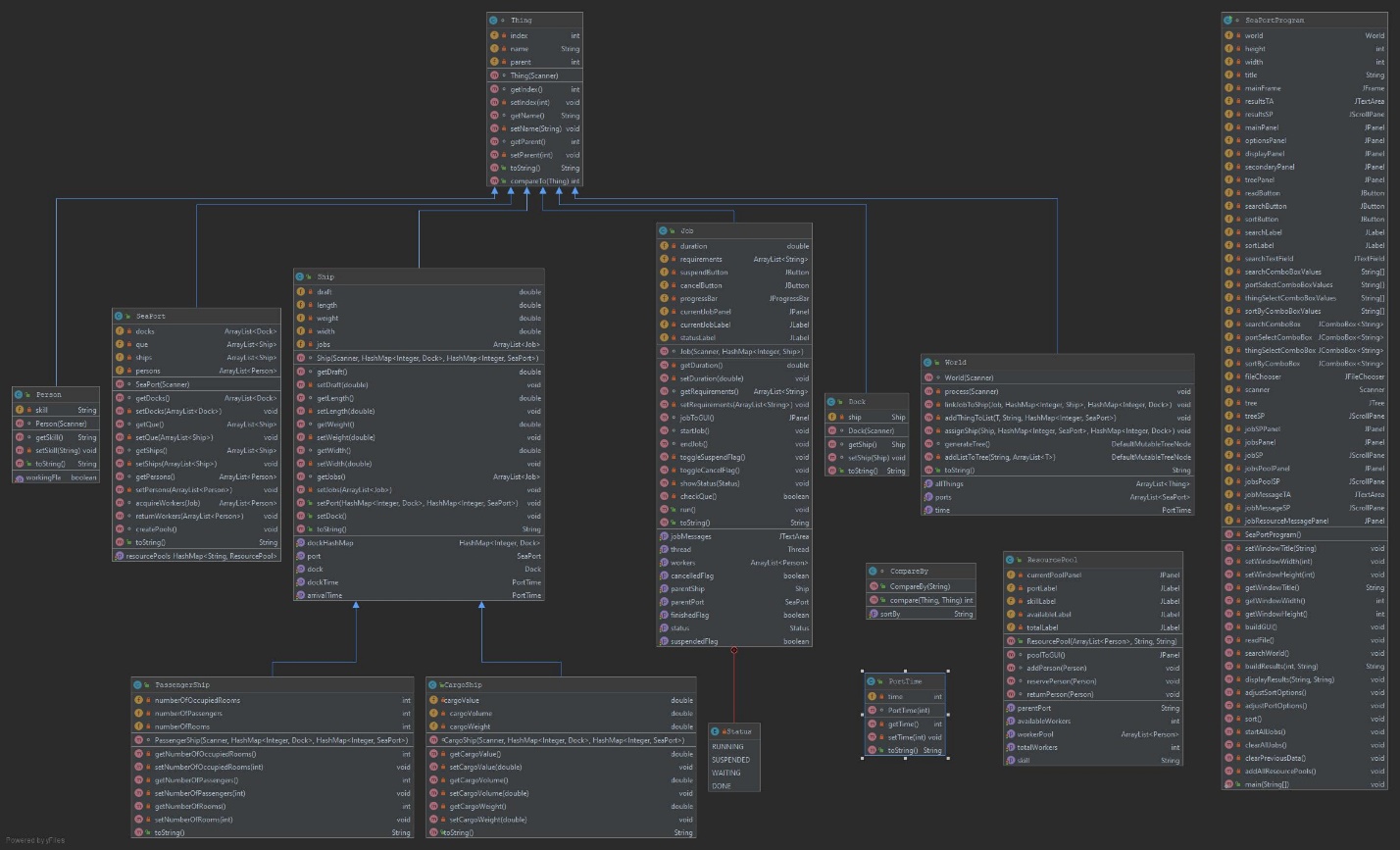
Antonio E. Ramirez

11 October 2019

Sea Port Project, Part 3

CMSC 335, 6381, University of Maryland University College

# UML Class Diagram



The UML.jpg file is included in the submission for better viewing of the class diagram.

# User’s Guide

The program is easy to operate. First, a properly formatted .txt file must be loaded using the “Read File” function at the bottom left within the GUI. Select a valid .txt file from the file chooser pop-up window. After a file has been loaded, multiple areas of the GUI will populate. First is the Job Status section located at the top of the GUI. This will display all your world’s jobs, their status, and contains the Suspend and Cancel buttons that pertain to each job. If a job is suspended, the status will change, and you will be able to resume the job at any time. If the cancel button is clicked, the status will also change but you will not be able to resume the job. The status for a cancelled job is the same as a completed job. The next section is below and to the left of the Job Status section, labeled Job Resources. In this section you will be able to see the resources of each port. In the first box is the skill, next is the owning port, then the number of currently available workers for that skill, and lastly is the total number of workers with that skill in the port. These numbers are dynamic and will update as workers are reserved and returned. Next to this section is the Job Messages section. In this section all pertinent messages regarding your jobs will be displayed. For example, if a port does not have any workers with a required skill to complete the job, a message explaining this will be displayed within thus section. This section is also dynamic and is constantly changing as jobs start, progress, and complete. There is a ton of valuable information that will be displayed in this section. All messages are saved for the current world and its jobs, so it can be looked at as more of a log. The messages update quite rapidly, so it can be hard to keep up with all the important information in real-time. Therefore, it is important that the text area saves all the current job messages. These messages are erased when loading a new world, so be sure the data is transferred prior to loading a new world. Next, in the lower left is the Tree section. This will populate with a JTree that organizes the loaded world. You are able to explore all things within your world through this section. It replaced the text area that was used in previous iterations of this project. The last section is the Results area that will display the results of sorting. Once all fields have properly populated, you can utilize the search function. Enter your search term into the text field, select the search parameter from the drop-down menu, and click search. A pop-up window will appear with your search results. If a valid .txt file has not been loaded, an error message will appear prompting you to load a world first. An error message will also appear prompting you to enter a valid search term. You will also see on the bottom of the main window sorting functionality. This also requires a valid .txt file to be loaded prior to sorting. Once a world has been initialized, you can select which port to sort through the dropdown menu. You can also choose to sort all ports. If there is only one port in the world then you will only have the “All Ports” option. After this, you can choose what aspect of the port you want to sort form the next dropdown menu. Currently, the “Que” option supports the most sorting functionality per the rubric. The rest of the options only support sorting by name. This leads into the final dropdown menu option, how you would like to sort the selected thing. Once you have selected all your sort parameters, click the sort button. Your sort results will appear in the “Results” text area. This will keep a log of all your sort queries. The sort results will be reset after loading a new .txt file and initializing a new world.

# Test Plan

Most of the work for this project part had already been completed when completing part 3. The biggest challenge will be implementing the resource pools. Looking into the project requirements, it seems each port will have multiple pools, one for each unique skill. Another consideration to keep in mind is tracking all each port’s pools. I tried to initially implement the pools as a list of lists within each instance of SeaPorts. There were numerous limitations and workarounds needed with this approach. The biggest challenge was transferring the data to be displayed in the GUI with this approach. I am sure the task could be accomplished with this approach, I was just having a hard time wrapping my head around it. In my research to find a more effective way to implement these resource pools, I found that creating a separate class would be the most effective. Within this class I could track the workers in the pool instance that had the skill. I could also track the amount of total and available workers more efficiently and send this data to the GUI by using a poolToGUI() method similar to the jobToGUI() method.

The other task was to display all relevant pool and job data to the GUI. Initial thought was to have a pop-up window appear with each message. This became unreasonable due to the insane amount of pop-up window’s required. I then considered a single summary pop-up window once all jobs had completed. This also seemed ineffective as the user would have to go back through a large summary and wait until the end to see if a job completed or did not have enough resources. It also didn’t do the user any good to see which resources were being taken and returned after the fact. It was finally decided that a dynamic TextArea would be the best approach to handle these messages. It was important that the older messages were not deleted with each update, so a user could go back through previous messages if needed. All messages are now able to be displayed in real-time and we can display a large amount of information. We can also go back and see the progression of jobs and the allocation of resources since the date remains if a new world is not loaded.

The other aspect beyond displaying job messages, is displaying pool data. As previously mentioned, this was done in a similar way to the way we display job progress. The data also needs to be dynamic, to show the currently available workers within the pool. This was not too difficult to implement after the ResourcePool class was implemented.

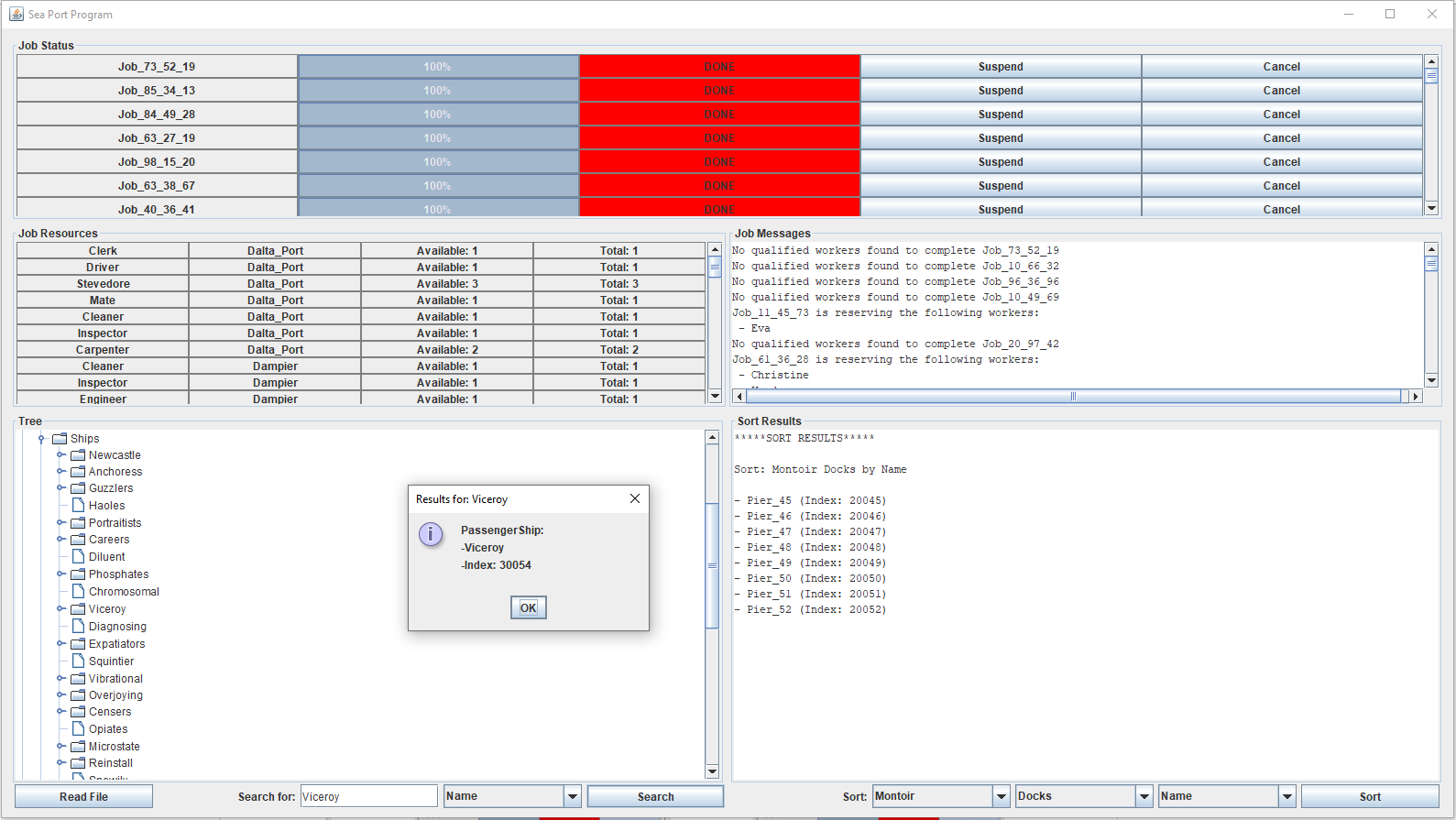
# Test Table

Like the previous project part, it is difficult to effectively show varied and effective test cases with real-time data such as the job progress, job messages, and resource pool availability. I will try to implement test cases that should intentionally fail and correct the issue to ensure the program works as intended. The focus of this testing will be on the part 4 objectives, as all previous objectives/functionality were not altered and will be shown to work in a general functionality test case. This test case will ensure that everything runs as intended, before delving deeper into the part 4 specific tests.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Number | Test Scenario | Input | Expected Output | Pass/Fail | Screenshot Set Number |
| 1 | General test case. Ensure all sections of the GUI function as intended, ensuring no previous functionality has been affected by any changes. | aSPae.txt  Sort: Montoir, Docks, Name  Search:  Viceroy, Name | All jobs should progress and complete, messages should update, resources should update, JTree should populate, sort and search should work as intended. | Pass | 1 |
| 2 | Check Suspend/ Cancel button function | aSPae.txt  Suspend:  Job\_98\_15\_20  Cancel:  Job\_40\_36\_41 | Job status change and progress pauses | Pass | 2 |
| 3 | Check sufficient resource functionality | aSPae.txt  Job\_73\_52\_19  Requirements:  Driver, Captain, Janitor  Missing:  Captain, Janitor  ADDED:  person Captain 50100 10003 captain  person Janitor 50101 10003 janitor | Job should not execute for first test since the port lacks the qualified workers.  Job should run and complete on the second run after adding the workers to the .txt file. | Pass | 3 |
| 4 | Check multiple of same skill job requirement functionality | aSPae.txt  Job\_73\_52\_19  Requirements:  Driver, Captain, Janitor  Added Requirement:  Driver | Job should not run on the first test since the job requires 2 drivers  Job should run on second test after another driver is added to the txt file | Pass | 4 |

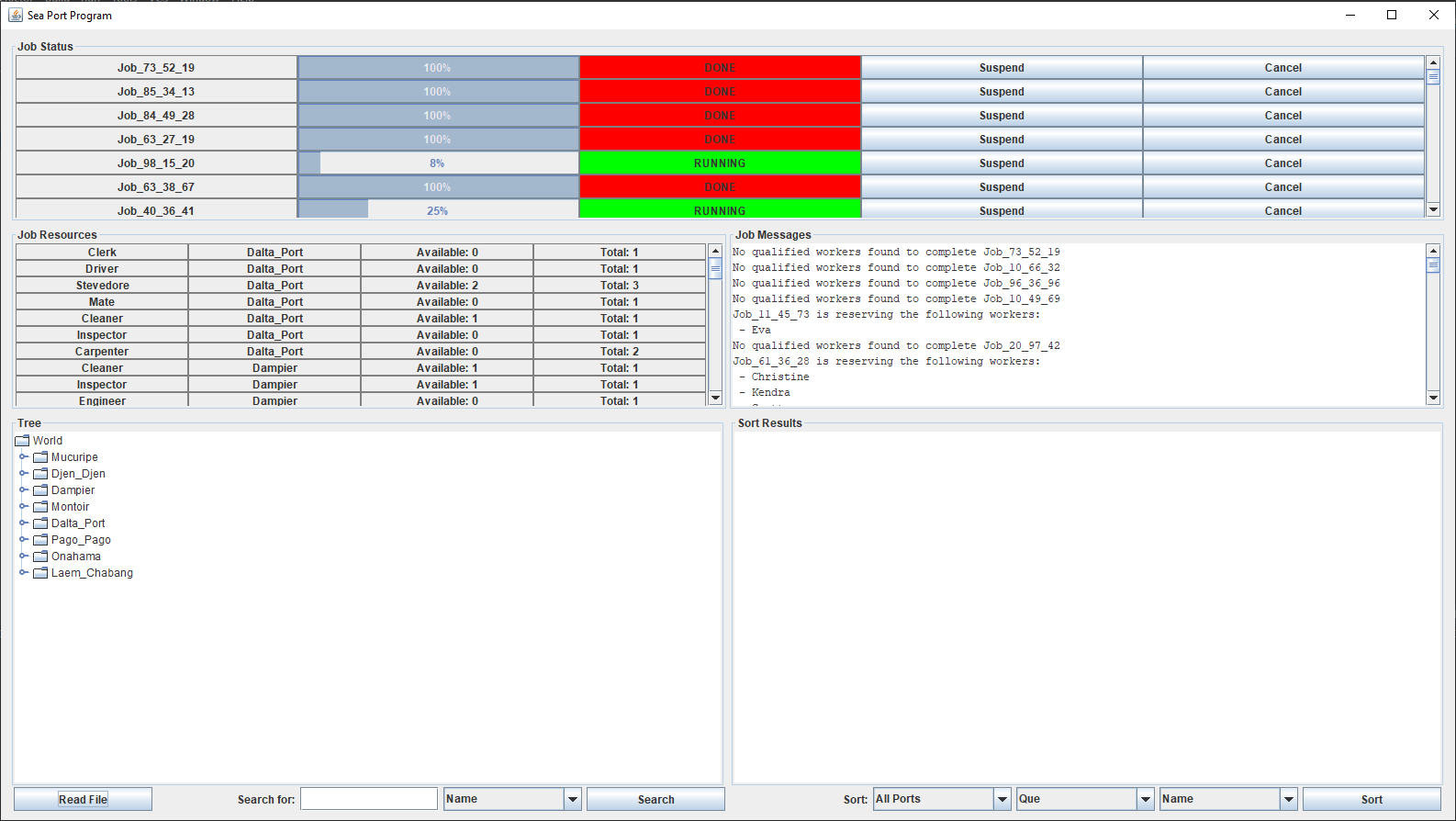
# Screenshots

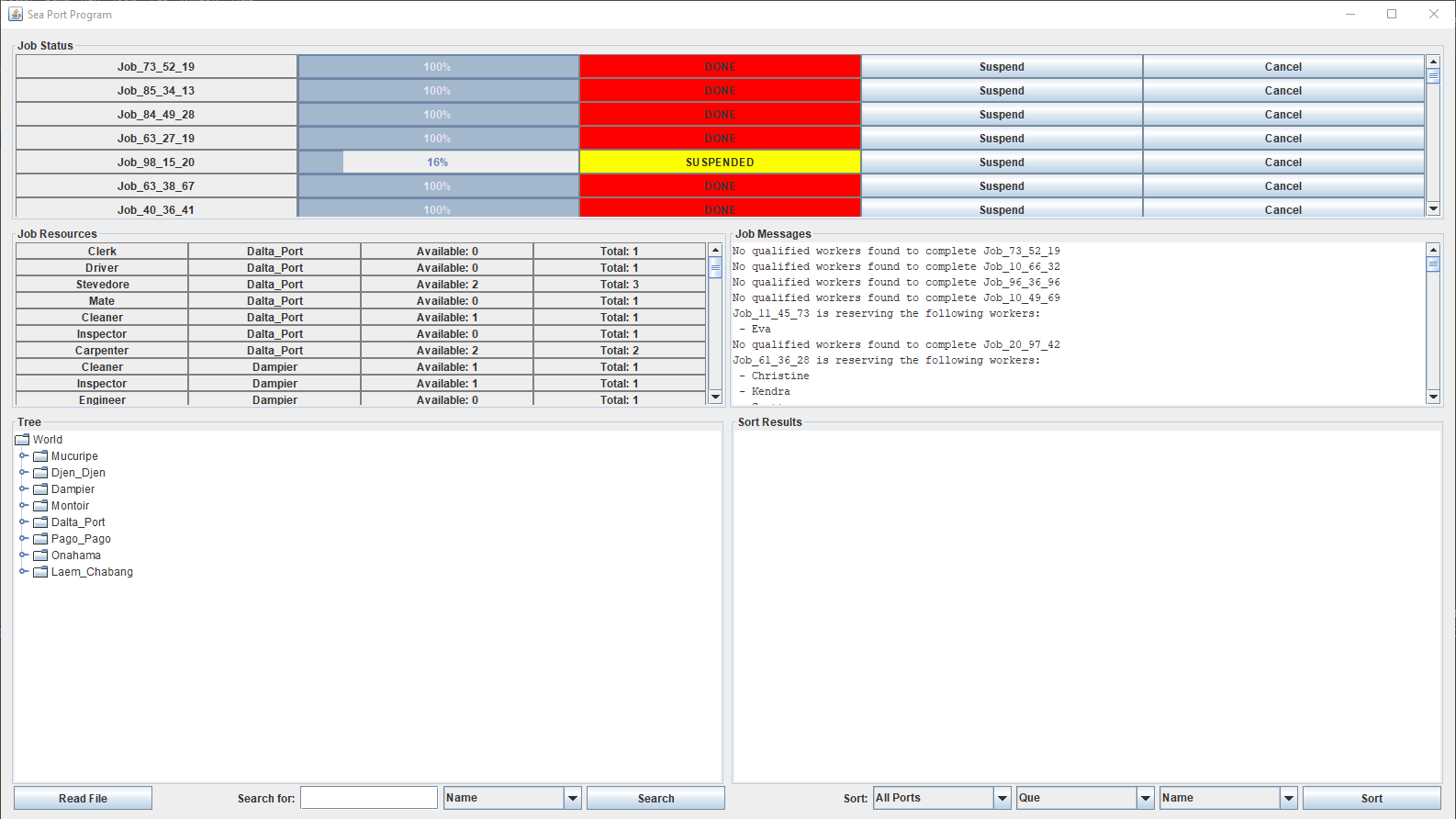
Screenshot 1

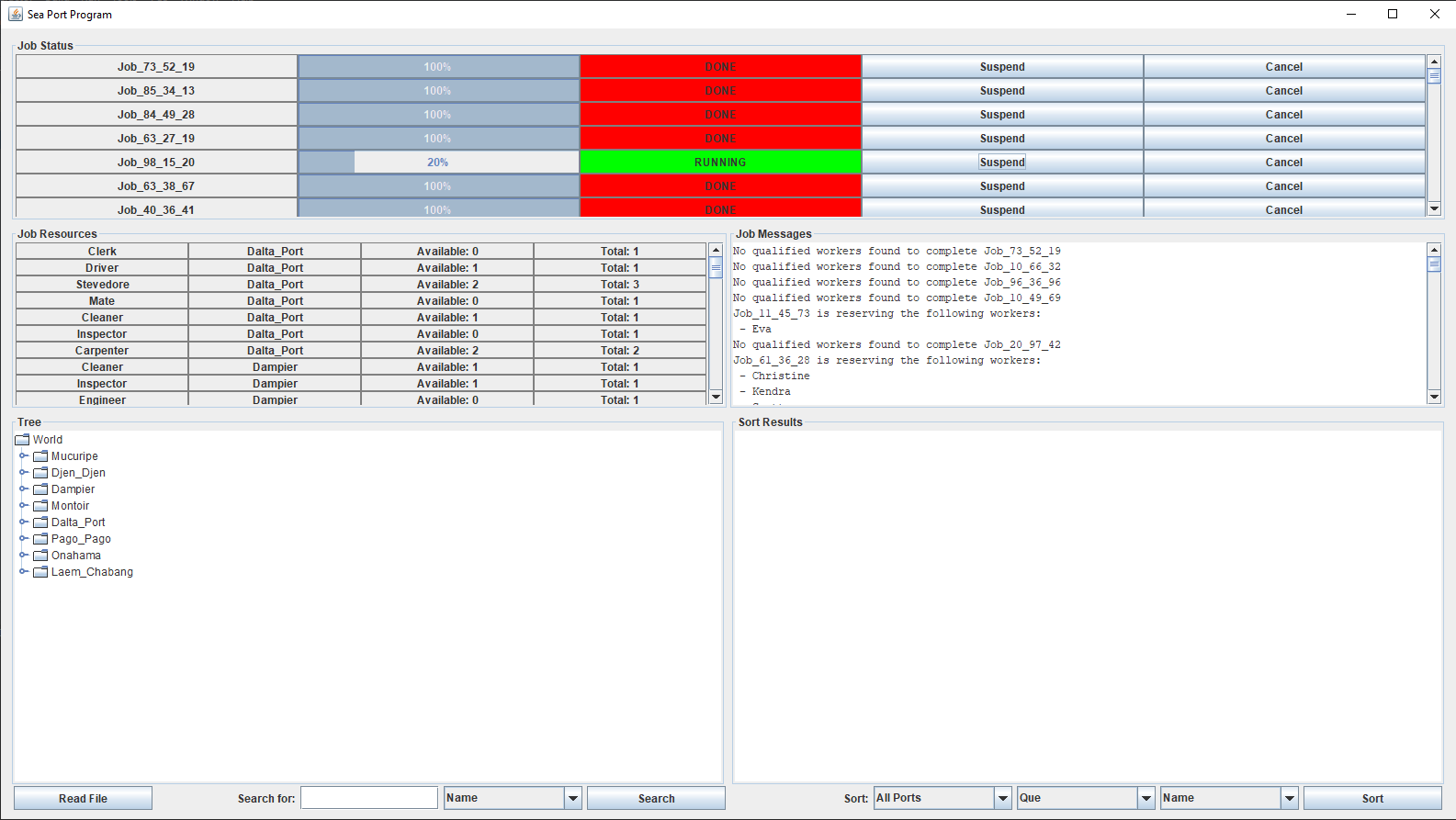


We can see the jobs displayed in the Job Status section progress, the Jobs Resources section displays each port’s pools and their available and total workers, the Jobs Messages section displays all necessary messages, the JTree populates with all world information and can be traversed, the sort function works as intended, and our search pop-up window still functions.

Screenshot set 2

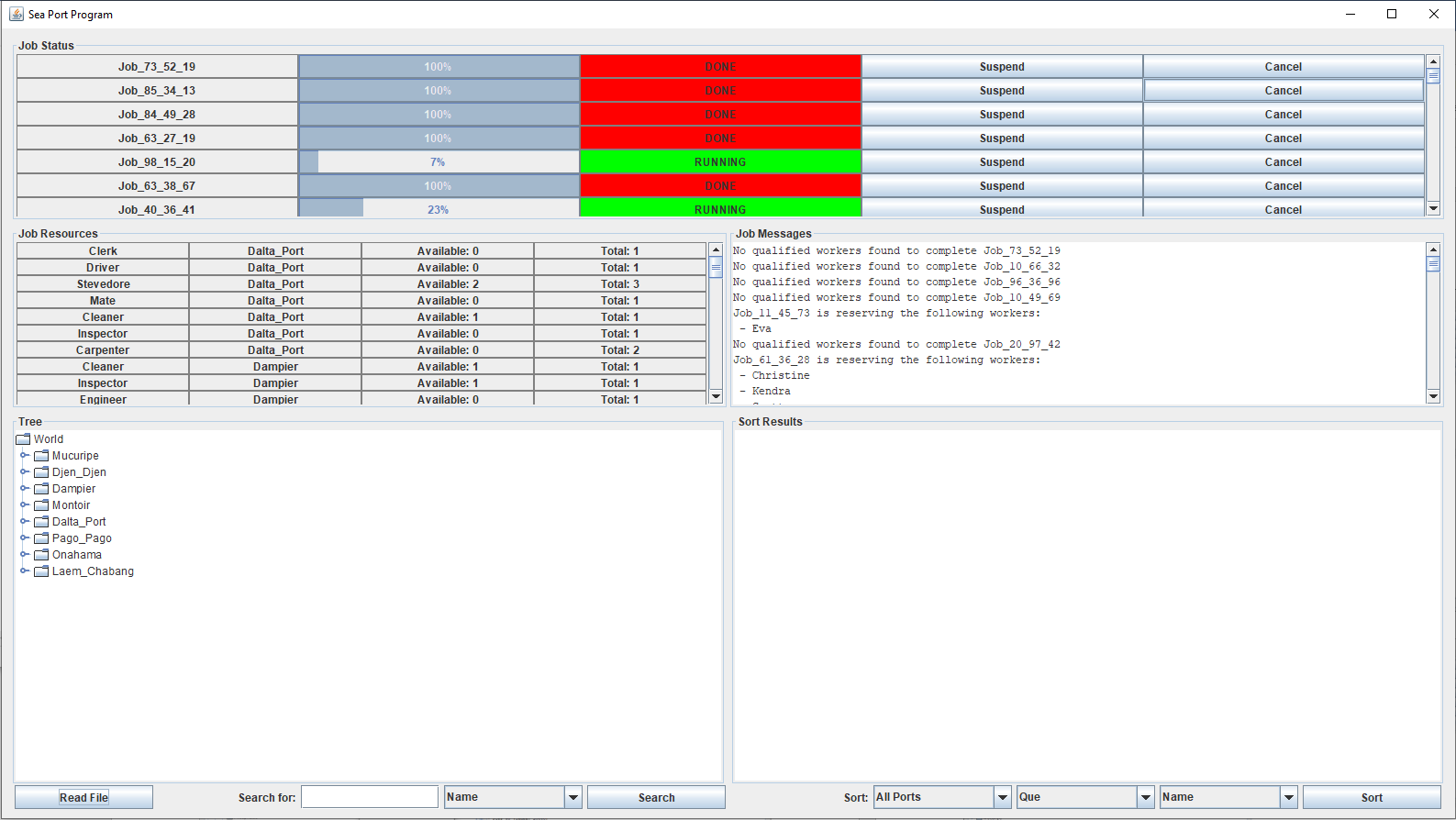


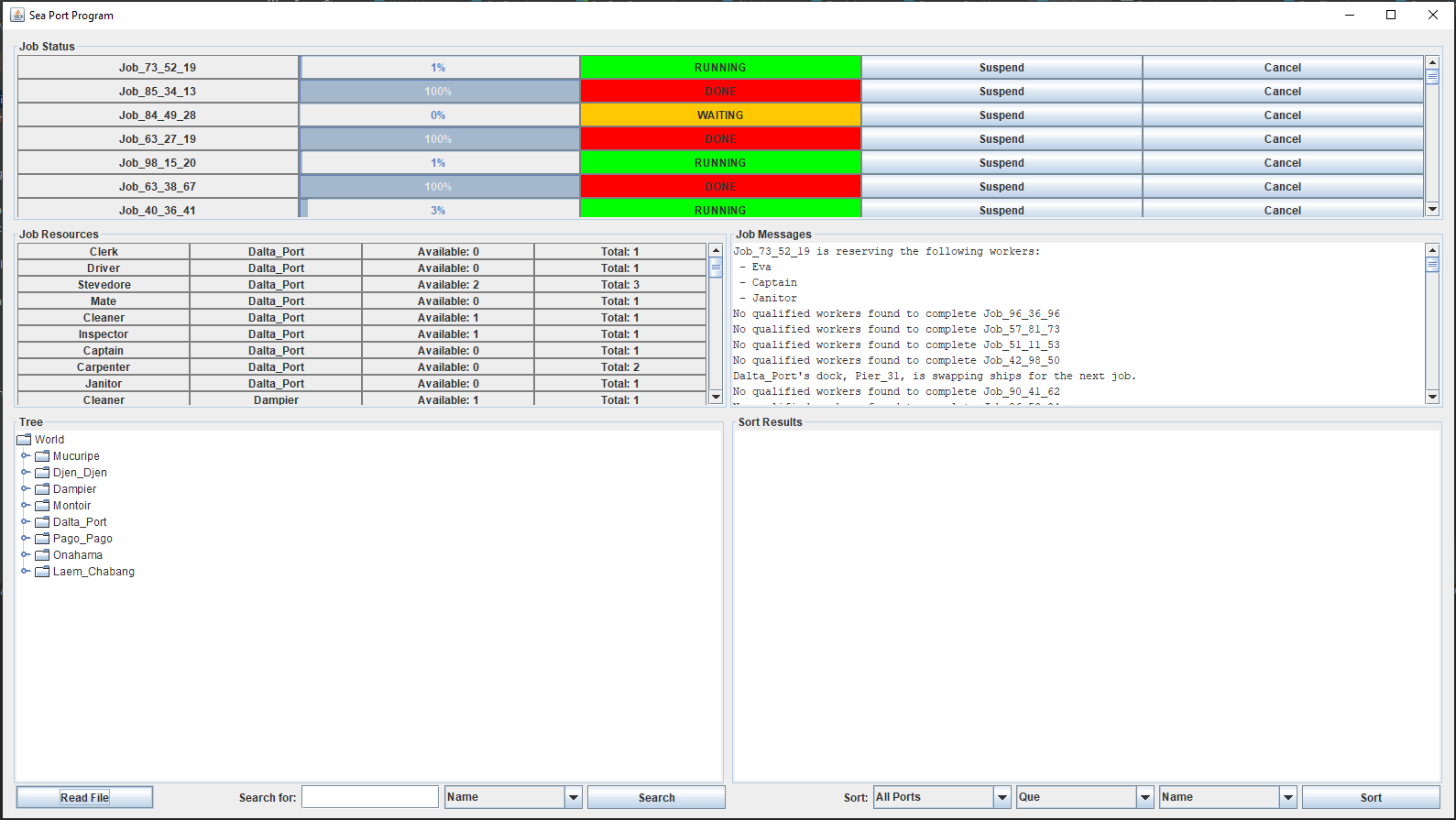




In the first screenshot we see both jobs running, as they should be. In the second screenshot, after clicking the Suspend button, the job’s status switches to the yellow SUSPENDED status. On the other job, after clicking the cancel button, we can see the status change to DONE and the progress jump to 100%. The job cannot be resumed after this. The third screenshot shows the suspended job being resumed by clocking the Suspend button again. The functionality of the 2 buttons has been confirmed.

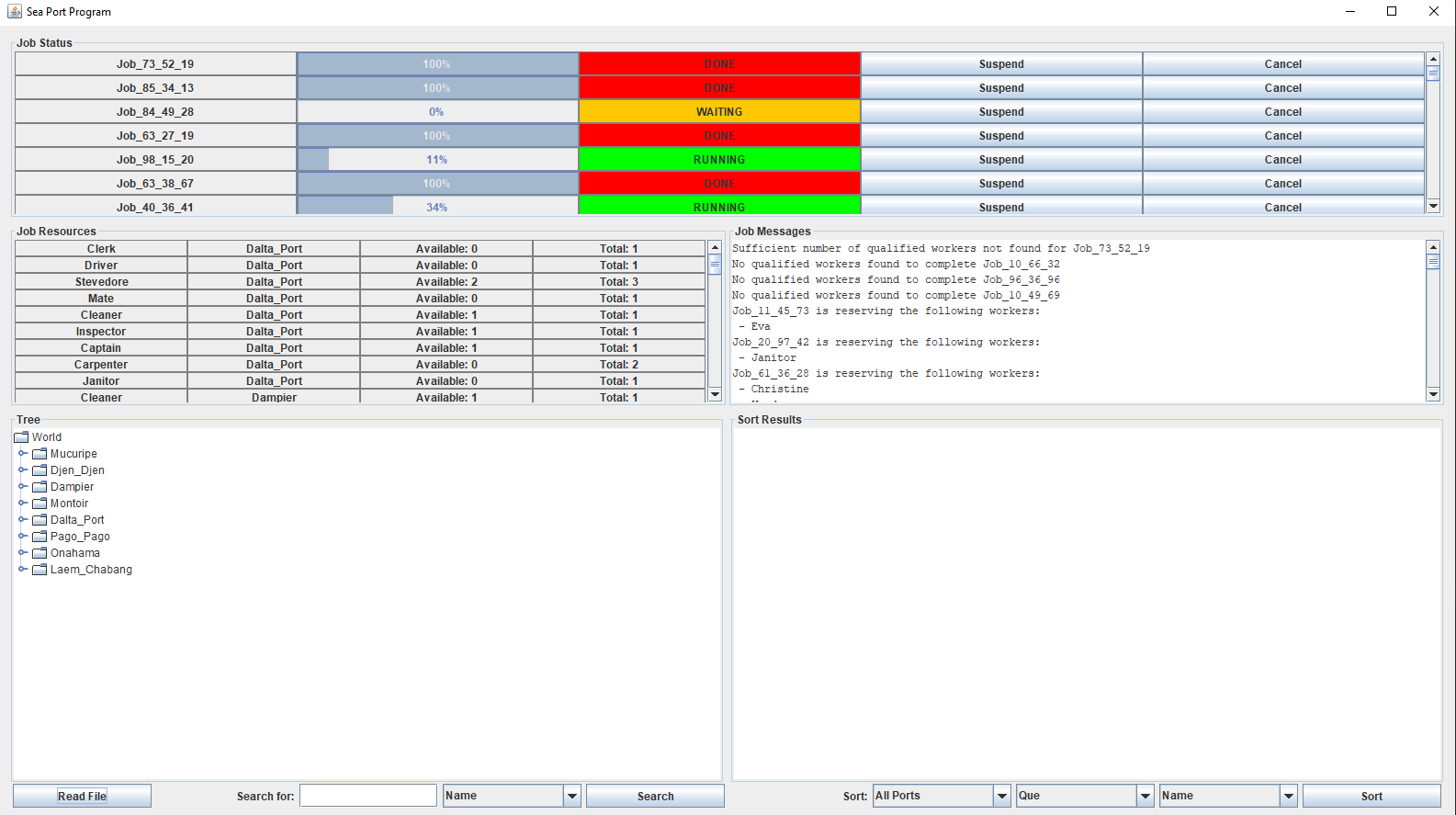
Screenshot set 3

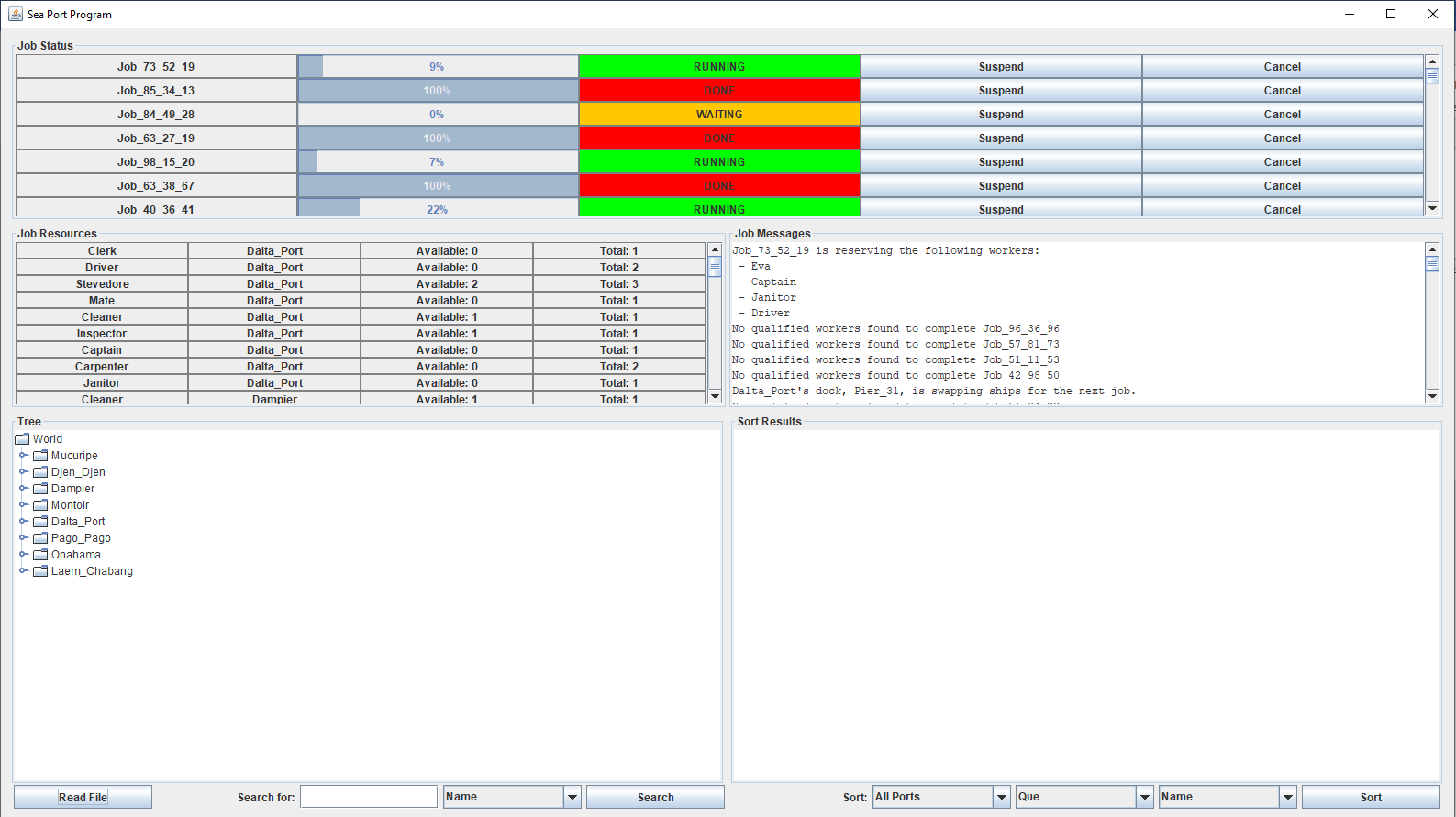




In the first screenshot we see that the first job is automatically displaying a status of DONE and a progress of 100%. This is further explained in the Job Messages section which explains no qualified workers could be found for the job in the first entry. After adding the captain and driver to the .txt input file, we can see the job is now running. This is confirmed in the Job Messages which says the job is reserving the workers in the first entry. The program passes the test case.

Screenshot set 4





For this test, we wanted to check that the program would handle multiple of the same skill requirement properly. This was done by adding an extra requirement for another driver to the first job. We can see in the first screenshot that the job does not run and a message saying a sufficient number of qualified workers could not be found. Once we add another driver to the .txt input file, the job runs, and the reserved workers can be seen in the first job message. The program passes the test case.

# Lessons Learned

This project taught me many new concepts with each part. For this part I was further tested by having to track resource pools and display them to the GUI. I was also challenged by having to find an effective way to display the individual job messages to the GUI. Dynamically updating the GUI to display the current pool information was similar to the way the job progress was displayed but was still difficult. This project exposed me to many new concepts within Java. The most difficult was the implementation of the job threads. Know where synchronized blocks needed to be implemented and what code needed to be in these blocks was difficult. Having these threads wait when resources were currently taken, and auto-complete when the resources did not exist was another challenging aspect in the project. Having the jobs check the required skill resource pools for the workers was how this problem was tackled. If the workers were available, signified by a working flag, then the job would reserve the worker, and the pool would update so the available worker count could reflect the new change. Overall, I learned a ton about resource management, especially when two different tasks require the same resource. I can see how the concepts form this project can be applied within computer science. Whether is two programs competing for the necessary memory access, or a contracting company tracking its resources across several jobs and their requirements. This project brought many, previously unthought-of, aspects of resource management to light.